Variations on the Visitor Pattern

Martin E. Nordberg III

Quintessoft Engineering, Inc.
3135 South State Street
Suite 108
Ann Arbor, MI 48108

July 22, 1996

Abstract

Two variations on the Visitor pattern are presented.

A *Default Visitor* provides default handlers for cases where the polymorphism of the hierarchy of elements can reduce the cost of code maintenance.

An *Extrinsic Visitor* implements double dispatch with run time type information instead of Accept( ) methods. With the same machinery it is possible to test the feasibility of a particular visit before performing its operation. The extrinsic visitor pattern provides several benefits in ease of software development in trade for poorer run time performance.

Interesting design tradeoffs between efficiency and maintainability determine the choice of the standard, default, or extrinsic visitor pattern in an application.

Introduction

The Visitor pattern is an increasingly widely used design pattern for the traversal of relatively fixed class hierarchies described in the book *Design Patterns*[1]. The presentation in this article relies on the reader's familiarity with the pattern as presented in that book.

There are a number of variations of the visitor pattern that prove useful for improving the traversal of a structure of "visitable" objects under certain circumstances. These are either specializations of the visitor pattern or additions to it.

The article describes two direct variations of the visitor pattern and then examines the design issues that influence their selection.

Default Visitor

Intent

Default Visitor adds another level of inheritance to the visitor pattern that provides a default implementation that takes advantage of the inheritance relationships in a polymorphic hierarchy of elements.

Motivation

One difficulty in applying the visitor pattern is modifying every concrete visitor whenever a new concrete element is added. For that matter, writing each VisitXxxx( ) method may be tedious if much of the behavior is the same for different classes derived from a common base class. If VisitXxxx( ) is often an exact copy of VisitYyyy( ), it seems a shame to use only the most primitive code reuse mechanism (cut and paste). The default visitor pattern provides default VisitXxxx( ) methods that move the tedious work to a single central location under these circumstances. Only those VisitXxxx( ) methods with unique behavior for a given concrete element must be written.

Consider, as a hypothetical example, a framework of classes with names like "Motor," "Sensor," "Conveyor," "SodaBottleConveyor," "ConveyorMotor," and "TemperatureSensor." We originally developed this framework for factory automation - real time control and management of a small factory. However, our management has recently decided to seek ISO9000 certification and wants us to add functionality to this system in the form of record keeping and reporting for service histories, quality problems, and the like. Because the new record keeping is secondary to, and separate from the original (and still primary) purpose of the code, we have decided to implement the new functionality with the Visitor pattern. We will add a small number of new attributes to the existing classes and a small number of new classes in the inheritance graph and will write concrete visitors for the new functions.

While the choice of the visitor pattern is well justified by the desire to separate new functionality from existing functionality, we find that we have difficulty in taking advantage of polymorphism in the framework hierarchy. For example, all motors are to be inspected every 90 days except machine tool motors that require 60 day inspection intervals. It seems a shame to design a class called BuildInspectionListVisitor that has a dozen repetitions of code like the following (for each different kind of motor)[1].

```
void BuildInspectionListVisitor::VisitConveyorMotor( ConveyorMotor& conveyorMotor )
  {
    if ( Now() - conveyorMotor.LastInspection() >= 90 )  AddToList( conveyorMotor );
  }
. . .
```

If this were the only visitor needed, we might take our lumps and get on with it, but there are many different visitors with different specializations, so we write the twelve repetitions once:

```
/*virtual*/ void DefaultVisitor::VisitMotor( Motor& motor )
  {}
void DefaultVisitor::VisitConveyorMotor( ConveyorMotor& conveyorMotor )
  {
    VisitMotor( conveyorMotor );
  }
. . .
```

and each common (default) case once:

```
class BuildInspectionListVisitor : public DefaultVisitor {
. . .

void BuildInspectionListVisitor::VisitMotor( Motor& motor )
  {
    if ( Now() - motor.LastInspection() >= 90 )  AddToList( motor );
  }
```

[1] The code is oversimplified, but the issues are not.

The new class, DefaultVisitor, from which the pattern derives its name, adds VisitXxxx methods for the abstract elements as well as the concrete elements and provides default implementations that traverse the inheritance graph. If we add a new class, WizBangRobotMotor, we still must modify DefaultVisitor but need to modify only a small subset of our concrete visitor classes.

Applicability

Apply the Default Visitor pattern when:

1. You would consider applying the Visitor pattern.

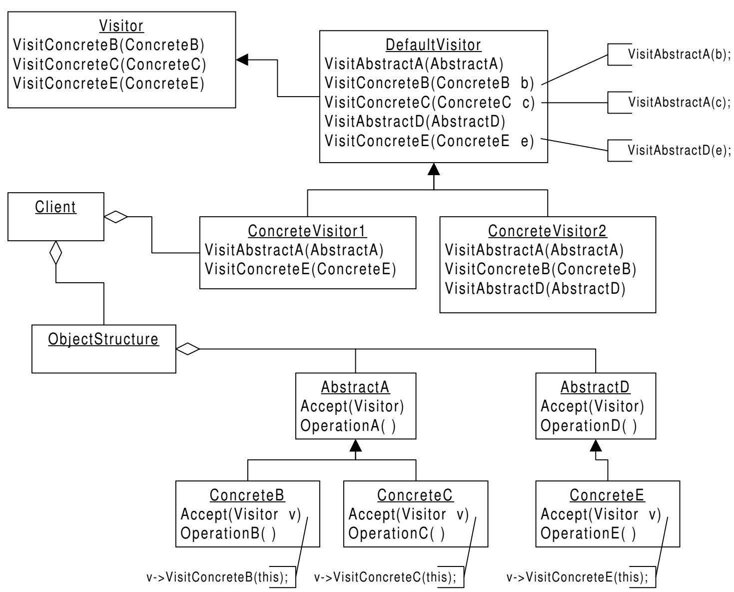
AND

2. Elements to be visited come from a small set of polymorphic class hierarchies.

AND

3. Several concrete visitors can employ default handlers for a small set of abstract elements rather than requiring a specific implementation for every distinct concrete element.

Structure



Participants

- **Visitor**

The Visitor class defines a VisitXxxx( ) method for each concrete class in the inheritance hierarchy of elements. As in the normal visitor class, the signature and name[2] of each VisitXxxx( ) method identify the concrete class of the element (Xxxx) being visited.

If all visitors in a system benefit from default handlers, then the Visitor class may be collapsed into the DefaultVisitor class. If not all visitors are properly polymorphic, it is desirable to leave the Visitor class separate in order to avoid errors in a standard concrete visitor when a concrete element is added to the hierarchy.

---

[2] See the footnote in the Gang-of-Four book[1] on page 337 and add to it the comment that if function overloading is used to name each method "Visit," there is no need to add a new method to Visitor and DefaultVisitor when a new concrete element class is added that needs no special treatment in any of the concrete visitors. However, this only delays (and probably magnifies) the maintenance headache until later when a new concrete visitor appears or a concrete element's behavior changes.

- **DefaultVisitor**

Derived from Visitor, a DefaultVisitor overrides each of the VisitXxxx( ) methods in Visitor and defines similar VisitXxxx( ) methods for each of the abstract classes in the inheritance hierarchy (or hierarchies) of elements. Each VisitXxxx( ) method, instead of being a do-nothing function, calls (with dynamic binding) the VisitXxxx( ) method for each base class of its argument.

- **ConcreteVisitor**

ConcreteVisitor implements a meaningful version of those VisitXxxx( ) methods that are relevant to the application. It may implement a combination of VisitXxxx( ) methods for concrete and abstract elements, taking advantage of the polymorphism in the element class hierarchy.

- **AbstractElement**

AbstractElement defines an Accept( ) method that receives a visitor as its argument. The Accept( ) method is the starting point for a client to cause a traversal of a structure of elements. Note that multiple abstract elements may exist in an inheritance hierarchy of elements.

- **ConcreteElement**

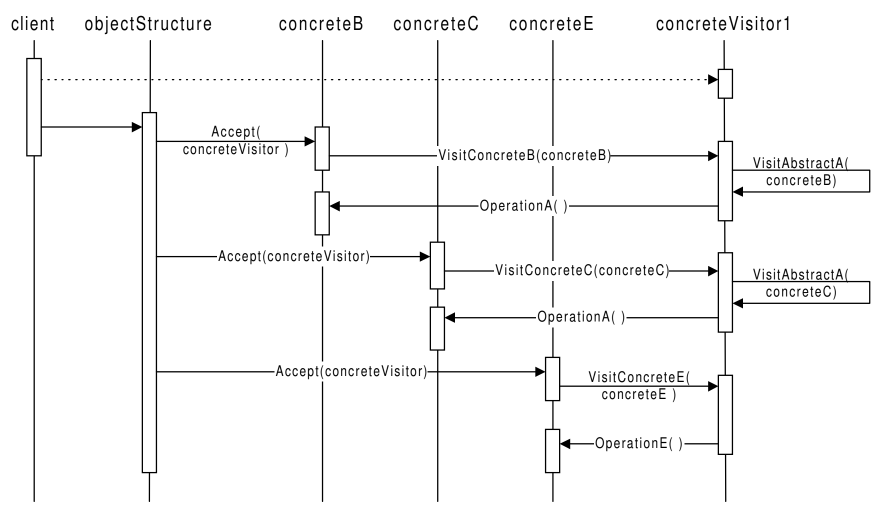
A concrete element implements a specific Accept( ) operation that calls the appropriate VisitXxxx( ) method of its visitor. For reduced complexity in this pattern (among other reasons[2]) each concrete element is preferably a leaf node of the inheritance hierarchy.

- **ObjectStructure**

The object structure is some collection of elements. An ObjectStructure provides the mechanism to traverse the collection or that mechanism is provided by either the elements themselves or helper functions in the base Visitor class.

Collaborations

With very little variation from the standard visitor pattern a client creates a ConcreteVisitor object and then traverses the object structure. All that changes is that some of the VisitXxxx( ) implementations may simply be the default ones. Below is (for example) the behavior of ConcreteVisitor1 from the structure diagram on the previous page:



The contribution of the object structure and the location for structure traversal may vary from one design to another. The client, the elements, or the visitor could optionally play some of the roles played by the object structure above.

Consequences

A default visitor makes it easier to add new ConcreteElement classes (compared to an ordinary visitor). As with the Visitor pattern, each new concrete element requires a new VisitXxxx( ) method in classes Visitor and DefaultVisitor. However, in cases where polymorphism is used to advantage (*i.e.* in cases where the Default Visitor pattern is used to advantage) the default implementation of the new VisitXxxx( ) method in DefaultVisitor will often be adequate for many of the different concrete visitor classes. Some of the concrete visitors may require no modification at all.

Though it is easier to add new leaf nodes to the element inheritance hierarchy, it is more difficult to change the inheritance hierarchy of abstract classes. Each such change must be carefully accounted for in the VisitXxxx( ) methods of class DefaultVisitor. Each concrete visitor must also be checked to make sure reliance on default implementations still makes sense.

The default visitor pattern requires more work in writing and maintaining class DefaultVisitor in return for less work in writing and maintaining each ConcreteVisitor class.

All of the remaining advantages and disadvantages of the default visitor pattern are much the same as described by Gamma, Helm, Johnson, and Vlissides[1] for the Visitor pattern.

Implementation

Implementation of the default visitor pattern is much the same as for the standard visitor pattern except for the VisitXxxx( ) members of the DefaultVisitor class. These methods should simply call the VisitXxxx( ) methods for the base classes of their argument. The diagram showing the pattern's structure includes notes suggesting implementations for the example VisitXxxx( ) methods of DefaultVisitor.

With care, even multiple and virtual inheritance can be handled. Each VisitConcreteXxxx( ) method should call the VisitXxxx( ) methods for its virtual base classes before calling the VisitXxxx( ) methods for its immediate, non-virtual base classes. The order of VisitXxxx( ) calls can in this way be made parallel to the order of constructor calls.

Known Uses

Default visitors appear widely in the C++ MetaCode Framework now under development by the author. For example, in this framework describing C++ code it is often sufficient to visit an Identifier rather than visit Classes and Functions and Enumerators and all the other types of identifiers in the framework hierarchy.

The author welcomes knowledge of other uses of this pattern.

Related Patterns

Default Visitor is a straightforward specialization of the Visitor pattern.

Extrinsic Visitor

Intent

The Extrinsic Visitor pattern trades the performance overhead of a small number of run time type tests for reduced complexity and coupling in the visitor and element classes of the pattern. The pattern also provides (in a single body of code) the ability to easily test for the feasibility of a visit operation before actually performing it.

Motivation

Consider a graphical system that incorporates widespread use of drag and drop operations. Imagine that in this system many different items can be dragged and dropped on one another, but only specific combinations of items are meaningful and the meaning varies with each different combination. The drop operation is thus a perfect candidate for double dispatch: the operation of dropping is likely to be outside the appropriate encapsulation for the items being dropped and the dynamic binding of the drop operation depends on the types of two items. A modern user interface provides greater feedback to the user during the drag operation by highlighting the item dropped on and changing the cursor style according to the potential drop operation. We need our visitor to provide not only Visit( ) operations (for the drop) but CanVisit( ) operations (for the drag).

The standard visitor pattern does not directly provide a mechanism for determining whether a particular combination of concrete visitor and concrete element is meaningful. Such a capability could be included by adding a parameter to the Accept( ) and VisitXxxx( ) methods to tell whether to perform the operation or merely test for its feasibility. Those two methods would then need to return a Boolean value indicating the feasibility. This approach is straightforward, but it only adds to the maintenance difficulties when adding new concrete element classes. Another approach with at least equal maintenance difficulties is to provide a separate, parallel visitor class to accomplish the test.

The extrinsic visitor pattern performs its dynamic dispatch with, in essence, a look up table by run time type of the visited element. The dispatch mechanism centralizes the distinction between performing an operation and testing for the feasibility of performing an operation. This approach thus trades run-time performance for easier code maintenance.

The extrinsic visitor pattern may also be appropriate, completely independent of the above arguments, when it is desired to add an operation to a hierarchy of classes that do not accept visitors and can not be modified to do so. The extrinsic visitor does not require Accept( ) methods in its elements; it requires only a common base class and run time type information. Even if Accept ( ) methods are feasible, the non-intrusiveness is also a significant benefit for code development since it eliminates the cyclic dependency between class Visitor and each of the concrete elements.

Applicability

Apply the Extrinsic Visitor pattern when:

1. You would consider the Visitor pattern for reasons of separation of dissimilar operations or easy addition of new operations.

AND

2. Elements to be visited all derive from a single (or a small number) of base classes.

AND

3. The number of distinct concrete elements visited by the typical concrete visitor is small.

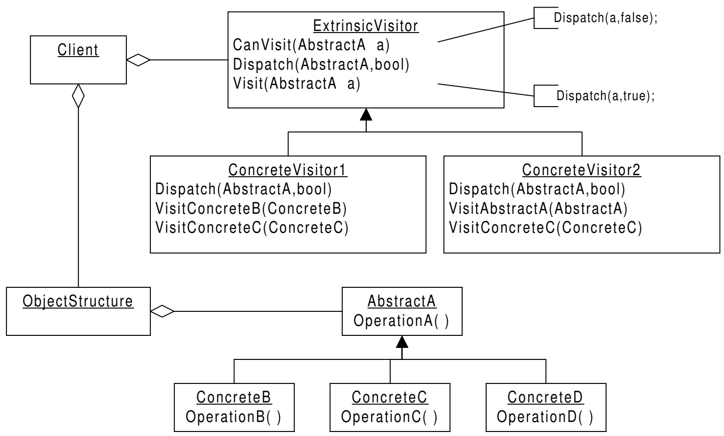
AND

4a. It is preferable to trade runtime performance for reduced intrusiveness of the Visitor pattern into the element hierarchy (no Accept( ) methods) and reduced complexity of the visitors themselves.

OR

4b. It is desirable to easily test whether a particular concrete visitor handles a particular concrete element without actually performing the operation on that element.

Structure



Participants

- **ExtrinsicVisitor**

This class is the root of the extrinsic visitor structure and provides two public functions to its clients: Visit(AbstractElement) and Boolean CanVisit(AbstractElement). Each of these depends on a single abstract method called Dispatch() which tests for the feasibility of operating on a particular concrete element and optionally performs the appropriate operation.

- **ConcreteVisitor**

Each concrete visitor class provides VisitXxxx( ) methods for relevant concrete element types and implements a corresponding version of Dispatch( ) which translates Visit( ) calls into VisitXxxx( ) calls or tests the feasibility for CanVisit( ) calls.

- **AbstractElement**

AbstractElement (AbstractA in the diagram) is the base class of the element structure. It participates in the extrinsic visitor pattern only to the extent that it provides run time type information. Concrete elements could be derived from a small set of abstract base elements, but ExtrinsicVisitor would need separate CanVisit(), Visit(), and Dispatch() methods for each abstract element.

- **ConcreteElement**

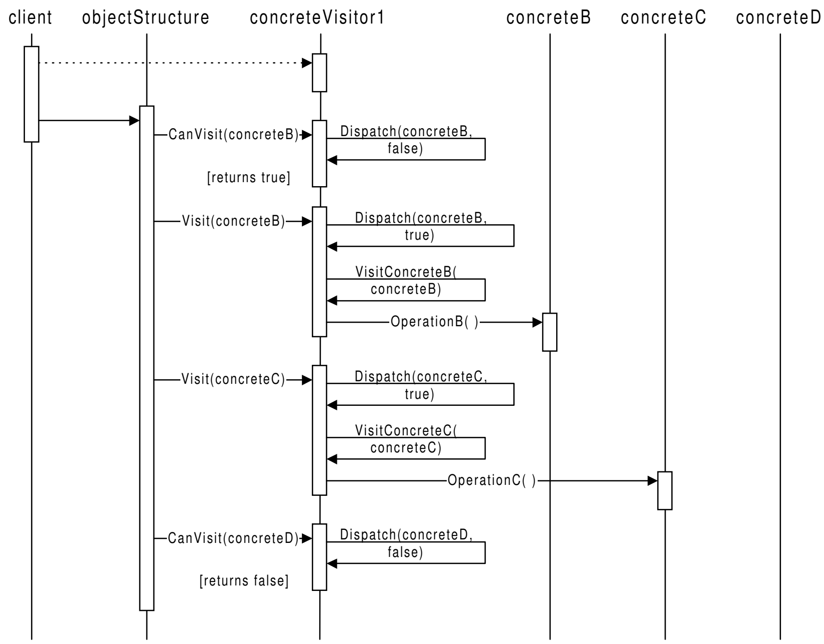
Concrete elements (*e.g.* ConcreteB and ConcreteC) are the items to be visited. They provide run time type information.

- **ObjectStructure**

As in the standard visitor pattern, the Object Structure is some container of items to be visited. In typical applications of the extrinsic visitor pattern it is common to visit only one item out of this structure rather than traversing the entire structure.

Collaborations

Because the extrinsic visitor pattern includes no Accept( ) methods, a "visit" starts with a direct call to the visitor. The visitor then dispatches the generic Visit( ) call to a VisitXxxx( ) call:



Because of this call graph the object structure must have knowledge of the ExtrinsicVisitor or the object structure may be bypassed altogether with the initial call to Visit( ) coming from the client. In appropriate applications of the extrinsic visitor pattern it is much more common for single elements to be visited rather than entire structures.

Consequences

The extrinsic visitor pattern implements dynamic binding through a relatively brute-force look up table that depends on run-time type of concrete elements. Depending on the implementation (see below) this approach may be considerably slower at runtime than the use of two virtual function tables in the standard visitor pattern.

The extrinsic visitor pattern requires a single abstract base class for its elements (or at least a small set of base classes). This may be undesirable for other design reasons (and has been a subject of much debate).

It is much easier to add new concrete classes to the extrinsic visitor pattern: only those concrete visitors that operate on objects of the new class need to be changed. The ExtrinsicVisitor base class needs modification only if a new base abstract element class is added.

The extrinsic visitor pattern also eliminates the need to recompile every concrete element when a new concrete element is added to the hierarchy. The standard visitor pattern has a cyclic dependency between each element and the abstract visitor. With Accept( ) methods gone the dependency is one way (visitors depend on elements) and the need for recompilation is reduced. In fact, not all concrete visitors need to be recompiled, only those that are modified to operate on objects of the new class.

None of the elements needs an Accept( ) method, but they all need to provide run time type information. In up-to-date C++ this is redundant with the single base class requirement (the abstract base class presumably has a virtual destructor). The requirement may be more difficult to satisfy in older or other environments.

Implementation

Most of the implementation details center around how to implement the Dispatch( ) methods in each concrete visitor.

One straightforward implementation is to provide a table of method pointers indexed by the type ID of each concrete element. If implemented as an array, the table emulates the virtual function table normally built by a compiler except that many entries will be null. If implemented as a map, the table provides a space-efficient representation of the dynamic binding needed but with a small performance overhead. Both of these methods require extra code (and more difficult code maintenance) to build and maintain the dispatch table. In C++ neither method readily allows for the advantages of the default visitor pattern of the previous section without careful hand coding of the table or extra run time type information providing an IsDerivedFrom( ) mechanism. For multiple or virtual inheritance in the element hierarchy the task may be impossible.

The author has found success with an implementation that trades even slower performance for relatively easy code creation and maintenance with the run time type information provided directly by the C++ language.

For example here is code to perform the dynamic dispatch needed by ConcreteVisitor1 in the diagram of the Structure section:

```
// perform the dispatch operations needed by ConcreteVisitor2 from the example earlier
bool ConcreteVisitor1::Dispatch( AbstractA& visitee ,
                                 bool do_the_call      )
  {
    ConcreteB* concreteB = dynamic_cast<ConcreteB*>( &visitee );
    if ( concreteB != 0 )
      {
        if ( do_the_call )
          VisitConcreteB( *concreteB );
        return true;
      }

    ConcreteC* concreteC = dynamic_cast<ConcreteC*>( &visitee );
    if ( concreteC != 0 )
      {
        if ( do_the_call )  // not just testing
          VisitConcreteC( *concreteC );
        return true;
      }

    return false;
  }
```

It is more convenient to extract the repeated structure of the above Dispatch function into a separate helper function. Each concrete visitor implements its dispatch mechanism by a series of calls to this global template function as in the following example:

```
// perform the dispatch operations needed by ConcreteVisitor2 from the example earlier
bool ConcreteVisitor2::Dispatch( AbstractA& visitee ,
                                 bool do_the_call       )
  {
    // below is the "dispatch table"
    return AbstractDoubleDispatch( this , &ConcreteVisitor2::VisitConcreteC ,
                                   visitee , do_the_call                     ) ||
           AbstractDoubleDispatch( this , &ConcreteVisitor2::AbstractA ,
                                   visitee , do_the_call                     );
  }
```

It may not be apparent from this small example, but the order of the calls to AbstractDoubleDispatch is significant both for correctness and efficiency. For example, if they were reversed above, VisitConcreteC would never be reached. In a larger system the calls could also be sub-ordered for efficiency, with more common cases placed first.

The global[3] template function AbstractDoubleDispatch performs the run time type test using the built in dynamic_cast operator.

```
// test for or perform double dispatch
template < class T_Visitor , class T_Visitee >
inline bool AbstractDoubleDispatch( T_Visitor* visitor ,
                                    void (T_Visitor::*visit)( T_Visitee& ) ,
                                    AbstractA& abstract_visitee ,
                                    bool do_the_call                       )
  {
    T_Visitee*  visitee = dynamic_cast<T_Visitee*>( &abstract_visitee );
    if ( visitee == 0 ) return false; // quit if can't dispatch

    if ( do_the_call ) (visitor->*visit)( *visitee );

    return true;
  }
```

This template function is unusual because the class and argument types of a method pointer distinguish instantiations of the template.

If the item to be visited is a fully concrete element rather than an abstract element, a more efficient dispatch mechanism is possible:

```
// test for or perform double dispatch on a concrete element
template < class T_Visitor , class T_Visitee >
inline bool ConcreteDoubleDispatch( T_Visitor* visitor ,
                                    void (T_Visitor::*visit)( T_Visitee& ) ,
                                    AbstractA& abstract_visitee ,
                                    bool do_the_call ,
                                    const type_info& visitee_type          )
  {
    if ( visitee_type == typeid(T_Visitee) )
      {
        if ( do_the_call )
          {
            // note: use static_cast here if no virtual inheritance
            T_Visitee*  visitee = dynamic_cast<Visitee*>( &abstract_visitee );
            (visitor->*visit)( *visitee );
          }
        return true;
      }
```

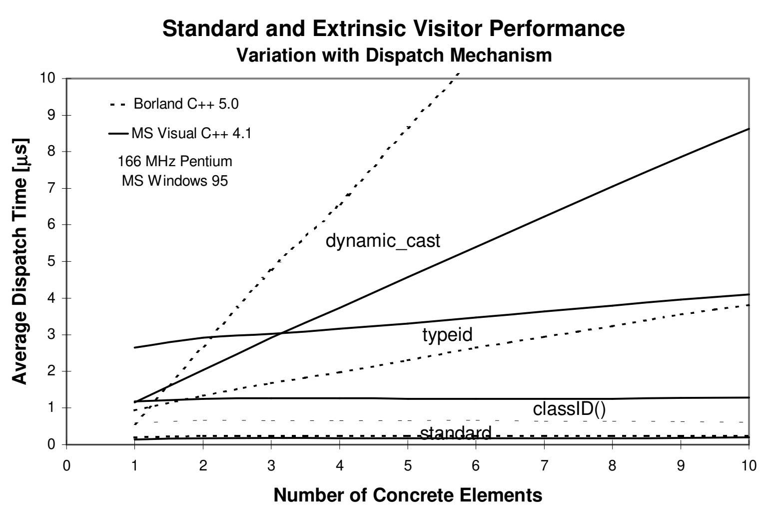
[3] It could be a member template function of class ExtrinsicVisitor given a compiler with that capability.

```
    return false;
  }

bool ConcreteVisitor2::Dispatch( AbstractA& visitee ,
                                 bool do_the_call       )
  {
    const type_info& visitee_type = typeid(visitee);

    return ConcreteDoubleDispatch( this , &ConcreteVisitor2::VisitConcreteC ,
                                   visitee , do_the_call , visitee_type       ) ||
           AbstractDoubleDispatch( this , &ConcreteVisitor2::AbstractA ,
                                   visitee , do_the_call                      );
  }
```

Though requiring only one dynamic_cast, ConcreteDoubleDispatch still requires searching through a list of options. If only concrete elements are to be dispatched, and we are allowed a minor intrusion into the element hierarchy to provide a ClassID( ) method for each element type, then a constant time implementation of the extrinsic visitor pattern is possible. For example:

```
bool ConcreteVisitor1::Dispatch( AbstractA& visitee ,
                                 bool do_the_call     )
  {
    switch ( visitee.ClassID() )
      {
      case ConcreteB::TheClassID:
        // note: use static_cast if no virtual inheritance
        if ( do_the_call ) VisitConcreteB( dynamic_cast<ConcreteB&>(visitee) );
        return true;
      case ConcreteC::TheClassID:
        if ( do_the_call ) VisitConcreteC( dynamic_cast<ConcreteC&>(visitee) );
        return true;
      default
        return false;
      }
  }
```

Note that ClassID( ) does not require the cyclic dependency that Accept( ) does and may be useful for other purposes. Following is a comparison of the three different implementations of extrinsic visitor under Windows 95 with two different compilers. These results are for a single level element hierarchy with a varying number of relevant concrete elements (visited with equal probability). Other compilers, platforms, or hierarchies will, of course, have different relative performance. The performance of the Standard Visitor is also shown for comparison.



Known Uses

The extrinsic visitor pattern sees use in the C++ MetaCode Framework now under development by the author. Design pattern implementations or simpler tool extensions (extrinsic visitors) may be dragged and dropped on individual classes or functions (elements) in a computer-aided software engineering tool. Extrinsic visitors in this CASE application form one basis for users of the tool to add functionality to it.

The author is eager to learn of other uses of this less intrusive though less efficient approach to double dispatch.

Related Patterns

Extrinsic Visitor is primarily a different implementation of the Visitor pattern.

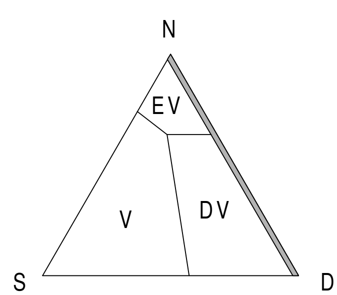
As a mechanism for implementing double dispatch the extrinsic visitor pattern extends in a straightforward manner to multiple dispatch, wherein the dynamic binding of an operation depends on the concrete type of a visitor and two or more different arguments ("visitees"). This is not true of the standard visitor pattern.

Choosing a Visitor Pattern Variation

The three pattern variations Visitor, Default Visitor, and Extrinsic Visitor are all solutions to the same problem that optimize different aspects of code performance and maintainability. An obvious question is when to choose each variation.

Materials scientists studying ceramics often draw triangle-shaped ternary phase diagrams. In such a diagram each vertex of the triangle represents 100% of one of the three components. Points within the triangle represent mixtures of the three materials. The relative nearness of a point to each vertex indicates the relative fraction of each component. For example, the center of the triangle represents a 1 : 1 : 1 mixture of the three components. Labeled regions within a ternary phase diagram represent phases, states of the material that are preferred at equilibrium (after cooling from a liquid).

The author proposes the following approximate "ternary phase diagram" for the three visitor variations based on the relative proportion of VisitXxxx( ) methods that are specific, default, and empty in a typical concrete visitor.



The three "phases" are EV (Extrinsic Visitor), DV(Default Visitor), and V(Visitor). The vertex labeled S indicates a concrete visitor in which every concrete element must have its own specific implementation of VisitXxxx( ). Vertex D represents a concrete visitor in which every

VisitXxxx( ) implementation can use the default implementation of class DefaultVisitor. Vertex N represents a concrete visitor in which all VisitXxxx( ) methods are no-op's or infeasible. The gray shaded region represents an impossible condition: every concrete visitor must have at least one specialized VisitXxxx( ) in order to be meaningfully called a concrete visitor. The region where the extrinsic visitor pattern is applicable is smallest on this diagram because of its run time overhead. This region would increase given a further requirement for a CanVisit( ) capability.

The regions of the graph are meant to be design guidelines under ideal circumstances. In the same way that ceramics can be super-cooled into a state different from their phase diagram, some designs may fall outside these guidelines because of other constraints. For example, an extrinsic visitor structure may be chosen because of the infeasibility of adding Accept( ) methods to an existing class hierarchy. In such a case the diagram may be a hint to optimize the implementation of the pattern in a different way. Perhaps the existing classes support a ClassID( ) mechanism and we can choose the more efficient implementation of Extrinsic Visitor.

The following table shows, in a general way, some of the tradeoffs that a designer makes when choosing from the three alternatives.

Pattern	Coupling Between Visitors and Elements	Run-Time Efficiency	Ease of Maintenance
Standard Visitor	Very High	High	Low
Default Visitor	High	High	Medium
Extrinsic Visitor	Low	Low	High

Some of these tradeoffs are common to many areas of software engineering.

Concluding Thoughts

Having completed this small catalog of variations on the Visitor design pattern, I now raise this question: is "variation on a pattern" the right name for them? Are these really separate design patterns, merely implementation details, or something in between? As evidence of the relevance of the question, neither of the patterns of this article stands on its own. Each assumes that the reader has already considered the visitor pattern and is looking to solve some problem of fit between that pattern and his or her application. The motivation and applicability sections, in particular, have been written this way. A better name for these patterns might be "sub-patterns" or "pattern implementations."

Actually, it is a general question in the development of software design patterns: how specific can they be and still be patterns? Do we need different names for patterns as they slide along a continuum of implementation detail from "write readable programs" to "module -> Accept(codeGenerationVisitor);"? Many researchers have developed metrics to measure object-oriented software interdependency, abstraction, reusability, domain independence, and so forth. Perhaps similar (though less formal) metrics will one day be applied to software design patterns.

The variations on the visitor pattern presented in this article have proved very useful to the author in the context of developing a computer-aided software engineering tool and are general enough to prove useful to developers in many other application areas.

Acknowledgments

The author acknowledges the constructive suggestions offered by primary reviewer Robert C. Martin, particularly concerning the performance comparison and coupling issues in the Extrinsic Visitor pattern. He also offered the current names for both pattern variations which had already changed several times in the author's own mind.

References

1. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, "Design Patterns - Elements of Reusable Object-Oriented Software," Addison-Wesley, Reading, MA, 1994.
2. Scott Meyers, "Code Reuse, Concrete Classes and Inheritance," *C++ Report 6*(6), July-August, 1994.